**Programs for Module 1**

**Q1. MULTI THREAD**

With the help of Multithreading concept, create two separate threads, one thread titled "primeThread" by extending the Thread class, and the other titled "fiboThread" by implementing Runnable interface.

"primeThread" will be responsible to print all the prime numbers from 1 to 100 in a regular interval of 0.25 seconds. "fiboThread" will be responsible to print fibonacci series of 20 numbers, in a regular interval of 0.5 seconds.

On executing this application, get the following info also.

1. Get the id,name and priority of the main thread
2. Change the name and priority of the main thread and print the same.
3. Print the thread group info of both the child threads
4. Use isAlive method to check the status of the childThread.

**Aim:**

The aim of this program is to demonstrate the use of multithreading by creating two separate threads, "primeThread" and "fiboThread." The "primeThread" is responsible for printing prime numbers from 1 to 100 at regular intervals, while the "fiboThread" prints a Fibonacci series of 20 numbers at regular intervals. Additionally, the program provides information about the main thread, changes its name and priority, and checks the status of the child threads.

**Algorithm:**

1. Create a class named `Prime` that extends the `Thread` class:

a. Initialize a boolean variable `isPrime` to `true`.

b. In the `run` method, loop through numbers from 2 to 99:

- Check if the number is prime by testing divisibility.

- If it's prime, print the number and sleep for 250 milliseconds.

c. Reset `isPrime` to `true` after each iteration.

2. Create a class named `Fibonacci` that implements the `Runnable` interface:

a. Initialize three integers: `n1` and `n2` as the initial Fibonacci values, and `MAX` as the maximum number of Fibonacci numbers to be printed.

b. In the `run` method, print the initial values of `n1` and `n2`.

c. Use a loop to generate the Fibonacci series up to the specified maximum:

- Calculate the next Fibonacci number (`n3`) based on `n1` and `n2`.

- Print `n3` and sleep for 500 milliseconds.

- Update `n1` and `n2` accordingly.

3. In the `PrimeandFibThread` class (the main class):

a. Get information about the main thread, such as its ID, name, and default priority.

b. Change the name and priority of the main thread and print the updated information.

c. Create an instance of the `Prime` class and start the "primeThread."

d. Create an instance of the `Fibonacci` class and a `Thread` for "fiboThread." Start "fiboThread."

e. Print information about the child threads.

f. Use the `isAlive` method to check the status of "primeThread."

**Program Explanation:**

- This program demonstrates multithreading by creating two separate threads: "primeThread" and "fiboThread."

- The `Prime` class extends the `Thread` class and is responsible for printing prime numbers from 1 to 100 at regular intervals. It uses a loop to check for prime numbers, and when a prime number is found, it is printed, and the thread sleeps for 250 milliseconds.

- The `Fibonacci` class implements the `Runnable` interface and is responsible for printing a Fibonacci series of 20 numbers at regular intervals. It initializes `n1` and `n2` as the initial Fibonacci values, and in a loop, it calculates the next Fibonacci number, prints it, and sleeps for 500 milliseconds.

- In the `PrimeandFibThread` class, the main thread is used to get information about the main thread itself, including its ID, name, and default priority. The program changes the name and priority of the main thread and prints the updated information.

- Two child threads, "primeThread" and "fiboThread," are created and started using instances of the `Prime` and `Fibonacci` classes, respectively. Information about the child threads is printed.

- The program uses the `isAlive` method to check if "primeThread" is still running.

- Overall, this program demonstrates the creation and management of multiple threads, showing how they can run concurrently and interact with each other.

**Program:**

**package** Scenario1;

**class** Prime **extends** Thread

{

**boolean** isPrime = **true**;

**public** **void** run() {

**for**(**int** num=2;num<100;num++)

{

**for**(**int** div=2;div<=num/2;div++)

{

**if**(num%div==0)

{

isPrime=**false**;

}

}

**if**(isPrime)

{

**try**

{

System.***out***.println("Prime :"+ num);

Thread.*sleep*(250);

}

**catch**(InterruptedException ie)

{

}

}

isPrime=**true**;

}

}

}

**class** Fibonacci **implements** Runnable

{

**int** n1 = 0, n2=1;

**final** **int** MAX =7;

**public** **void** run()

{

System.***out***.println("Fib: "+n1);

System.***out***.println("Fib: "+n2);

**for**(**int** i=2;i<MAX;i++){ **int** n3 = n1+n2;

**try**{

System.***out***.println("Fib: "+ n3);

Thread.*sleep*(500);

}**catch**(InterruptedException ie){

}

n1=n2; n2=n3;

}

}

}

**public** **class** PrimeandFibThread {

**public** **static** **void** main(String[] args) {

//get the name and Id of the main thread

Thread mainThread = Thread.*currentThread*();

System.***out***.println("Id of Main Thread: "+ mainThread.getId());

System.***out***.println("Name of main Thread: "+ mainThread.getName());

System.***out***.println("Default Priority of Main Thread: "+ mainThread.getPriority());

//Change the name and priority of the main thread and print the same.

mainThread.setName("ThreadMainClass");

mainThread.setPriority(Thread.***MIN\_PRIORITY***);

System.***out***.println("Name of main Thread after changing: "+mainThread.getName()); System.***out***.println("Priority of main thread after changing: "+mainThread.getPriority());

//creating a child thread "primeThread"

Prime prime= **new** Prime();

prime.setName("primeThread");

prime.start();

//creating a child thread "fiboThread"

Fibonacci fibonacci = **new** Fibonacci();

Thread fib = **new** Thread(fibonacci,"fiboThread"); fib.start();

//Thread grouop of the child-Thread

System.***out***.println("child-Thread1: "+ prime);

System.***out***.println("child-Thread2: "+fib);

//checking with isAlive() method

System.***out***.println("primeThread is Alive??: "+ prime.isAlive());

}

}

**Output:**

Id of Main Thread: 1

Name of main Thread: main

Default Priority of Main Thread: 5

Name of main Thread after changing: ThreadMainClass

Priority of main thread after changing: 1

Prime :2

child-Thread1: Thread[primeThread,1,main]

child-Thread2: Thread[fiboThread,1,main]

primeThread is Alive??: true

Fib: 0

Fib: 1

Fib: 1

Prime :3

Fib: 2

Prime :5

Prime :7

Fib: 3

Prime :11

Prime :13

Fib: 5

Prime :17

Prime :19

Fib: 8

Prime :23

Prime :29

Prime :31

Prime :37

Prime :41

Prime :43

Prime :47

Prime :53

Prime :59

Prime :61

Prime :67

Prime :71

Prime :73

Prime :79

Prime :83

Prime :89

Prime :97

**Q2. MULTI THREAD**

Design and develop a program for create two separate threads, one thread titled "PrimeThread" by extending the Thread class, and the other titled "PalindromeThread" by implementing Runnable interface. Palindrome thread has to display the palindrome of numbers between 100 to 999. Prime thread has to display all the prime numbers within 100. Display the list of palindromes and PrimeNumbers. Check the existence of threads using isAlive method. Make “PRIME” thread to wait for half a second till other thread completes its task.

On executing this application, get the following info also.

1. Get the id,name and priority of the main thread
2. Change the name and priority of the main thread and print the same.
3. Print the thread group info of both the child threads
4. Use isAlive method to check the status of the childThread.

**Aim:**

The aim of this Java program is to create and manage two separate threads, "Prime" and "Palindrome." The "Prime" thread is responsible for finding and displaying prime numbers within the range of 100 to 999, and the "Palindrome" thread is responsible for finding and displaying palindrome numbers within the same range. The program also demonstrates the use of thread properties such as thread names, priorities, and the `isAlive()` method.

**Algorithm:**

1. The program starts by obtaining information about the main thread, such as its ID, name, and priority. It then changes the name and priority of the main thread.

2. The "Prime" thread is created as a class that extends `Thread`. It is responsible for finding and displaying prime numbers in the range of 100 to 999. This thread uses a nested loop to check for prime numbers, and when a prime number is found, it is printed with a sleep of 500 milliseconds between each prime number.

3. The "Palindrome" thread is created as a class that implements `Runnable`. It is responsible for finding and displaying palindrome numbers within the same range. This thread uses a helper method `isPalindrome()` to check for palindromes, and when a palindrome is found, it is printed.

4. The main program creates and starts both the "Prime" and "Palindrome" threads.

5. The program displays information about the child threads, such as their names and IDs.

6. The `isAlive()` method is used to check if the "Prime" thread is still alive, and the result is printed.

**Program Explanation:**

- The program starts by obtaining information about the main thread, such as its ID, name, and priority. It changes the name to "ThreadMainClass" and sets the priority to the minimum priority (MIN\_PRIORITY).

- The "Prime" thread is created as an extension of the `Thread` class. It checks for prime numbers within the range of 100 to 999 and prints them with a 500-millisecond delay between each prime number.

- The "Palindrome" thread is created as an instance of a class that implements `Runnable`. It checks for palindrome numbers within the same range and prints them.

- The main program creates and starts both the "Prime" and "Palindrome" threads.

- The program displays information about the child threads, including their names and IDs.

- It uses the `isAlive()` method to check if the "Prime" thread is still alive and prints the result.

The program demonstrates the concurrent execution of two threads for finding prime and palindrome numbers and showcases the usage of thread properties and methods in Java multithreading.

**Program:**

**package** Scenario1\_1;

**class** Prime **extends** Thread{

**boolean** isPrime = **true**;

**public** **void** run()

{

**for**(**int** num=2;num<100;num++)

{

**for**(**int** div=2;div<=num/2;div++)

{

**if**(num%div==0)

{

isPrime=**false**;

}

}

**if**(isPrime)

{

**try**

{

System.***out***.println("Prime :"+ num); Thread.*sleep*(500);

}

**catch**(InterruptedException ie)

{

}

}

isPrime=**true**;

}

}

}

**class** Palindrome **implements** Runnable

{

**public** **void** run()

{

**for** (**int** i = 100; i <= 999; i++) {

**if** (isPalindrome(i)) {

System.***out***.println("Palindrome: " + i);

}

}

}

**private** **boolean** isPalindrome(**int** number) {

// Reverse the number

**int** reversedNumber = 0;

**int** originalNumber=number;

**while** (number > 0) {

**int** remainder = number % 10;

reversedNumber = reversedNumber \* 10 + remainder;

number /= 10;

}

// Check if the reversed number is equal to the original number

**return** originalNumber == reversedNumber;

}

}

**public** **class** PlainandPrime {

**public** **static** **void** main(String[] args) {

//get the name and Id of the main thread

Thread mainThread = Thread.*currentThread*();

System.***out***.println("Id of Main Thread: "+ mainThread.getId());

System.***out***.println("Name of main Thread: "+ mainThread.getName());

System.***out***.println("Default Priority of Main Thread: "+ mainThread.getPriority());

//Change the name and priority of the main thread and print the same.

mainThread.setName("ThreadMainClass");

mainThread.setPriority(Thread.***MIN\_PRIORITY***);

System.***out***.println("Name of main Thread after changing: "+mainThread.getName()); System.***out***.println("Priority of main thread after changing: "+mainThread.getPriority());

//creating a child thread "primeThread"

Prime prime= **new** Prime();

prime.setName("primeThread");

prime.start();

//creating a child thread "fiboThread"

Palindrome palindrome = **new** Palindrome();

Thread pali = **new** Thread(palindrome,"PalindromeThread");

pali.start();

//Thread grouop of the child-Thread

System.***out***.println("child-Thread1: "+ prime);

System.***out***.println("child-Thread2: "+pali);

//checking with isAlive() method

System.***out***.println("primeThread is Alive??: "+ prime.isAlive());

}

}

**Output:**

Id of Main Thread: 1

Name of main Thread: main

Default Priority of Main Thread: 5

Name of main Thread after changing: ThreadMainClass

Priority of main thread after changing: 1

Prime :2

child-Thread1: Thread[primeThread,1,main]

child-Thread2: Thread[PalindromeThread,1,main]

primeThread is Alive??: true

Palindrome: 101

Palindrome: 111

Palindrome: 121

Palindrome: 131

Palindrome: 141

Palindrome: 151

Palindrome: 161

Palindrome: 171

Palindrome: 181

Palindrome: 191

Palindrome: 202

Palindrome: 212

Palindrome: 222

Palindrome: 232

Palindrome: 242

Palindrome: 252

Palindrome: 262

Palindrome: 272

Palindrome: 282

Palindrome: 292

Palindrome: 303

Palindrome: 313

Palindrome: 323

Palindrome: 333

Palindrome: 343

Palindrome: 353

Palindrome: 363

Palindrome: 373

Palindrome: 383

Palindrome: 393

Palindrome: 404

Palindrome: 414

Palindrome: 424

Palindrome: 434

Palindrome: 444

Palindrome: 454

Palindrome: 464

Palindrome: 474

Palindrome: 484

Palindrome: 494

Palindrome: 505

Palindrome: 515

Palindrome: 525

Palindrome: 535

Palindrome: 545

Palindrome: 555

Palindrome: 565

Palindrome: 575

Palindrome: 585

Palindrome: 595

Palindrome: 606

Palindrome: 616

Palindrome: 626

Palindrome: 636

Palindrome: 646

Palindrome: 656

Palindrome: 666

Palindrome: 676

Palindrome: 686

Palindrome: 696

Palindrome: 707

Palindrome: 717

Palindrome: 727

Palindrome: 737

Palindrome: 747

Palindrome: 757

Palindrome: 767

Palindrome: 777

Palindrome: 787

Palindrome: 797

Palindrome: 808

Palindrome: 818

Palindrome: 828

Palindrome: 838

Palindrome: 848

Palindrome: 858

Palindrome: 868

Palindrome: 878

Palindrome: 888

Palindrome: 898

Palindrome: 909

Palindrome: 919

Palindrome: 929

Palindrome: 939

Palindrome: 949

Palindrome: 959

Palindrome: 969

Palindrome: 979

Palindrome: 989

Palindrome: 999

Prime :3

Prime :5

Prime :7

Prime :11

Prime :13

Prime :17

Prime :19

Prime :23

Prime :29

Prime :31

Prime :37

Prime :41

Prime :43

Prime :47

Prime :53

Prime :59

Prime :61

Prime :67

Prime :71

Prime :73

Prime :79

Prime :83

Prime :89

Prime :97

**Q3. SYNCHRONIZATION**

John is participating a general quiz, there he got a question to display “n” numbers in reverse to finish the answer. Develop a java program to help john to this problem using static synchronization in threads.

**Aim:**

The aim of the Java program is to create multiple threads that access a synchronized static method `display` of the `exam1` class. This program allows the user to input an integer 'n', and then each thread counts down from 'n' to 1 with a 1-second delay between each count. The program demonstrates the use of synchronization in a static method.

**Algorithm:**

1. Create a class `exam1` with a synchronized static method `display`. This method takes input for an integer 'n' and then counts down from 'n' to 1, printing each value with a 1-second delay between each count.

2. Create a class `Thrs` that extends the `Thread` class. In its `run` method, call the synchronized `display` method from the `exam1` class.

3. In the `quiz` class (the main class), create three `Thrs` objects, representing three threads.

4. Start all three threads by invoking their `start` methods.

**Program Explanation:**

- The program starts by creating a class named `exam1` with a synchronized static method `display`. This method reads an integer 'n' from the user and counts down from 'n' to 1, printing each value with a 1-second delay between each count.

- The `Thrs` class is defined, extending the `Thread` class. In its `run` method, it calls the synchronized `display` method from the `exam1` class. This means multiple threads can access the method, but only one thread can execute it at a time due to the `synchronized` keyword.

- In the `quiz` class (the main class), three `Thrs` objects, namely `t1`, `t2`, and `t3`, are created.

- The `start` method is called on each thread, which initiates the execution of the `run` method in each thread.

- The program demonstrates how synchronization works with a static method to ensure that only one thread at a time can execute the `display` method, preventing concurrent access that could lead to unexpected behavior.

Overall, this program showcases the use of synchronized methods and multi-threading in Java.

**Program:**

**package** Scenario2;

**import** java.util.Scanner;

**class** exam1 {

**synchronized** **static** **void** display(){

//Thread t=Thread.currentThread();

System.***out***.println("enter the value of n");

Scanner sc=**new** Scanner(System.***in***);

**int** n=sc.nextInt();

**for**(**int** i=n; i>=1; i--){

**try**{

Thread.*sleep*(1000);

System.***out***.println("The value of i is: "+i);

}

**catch**(Exception e){

}

}

}

}

**class** Thrs **extends** Thread {

**public** **void** run(){

exam1.*display*();

}

}

**public** **class** quiz {

**public** **static** **void** main(String[] args) {

Thrs t1=**new** Thrs();

Thrs t2=**new** Thrs();

Thrs t3=**new** Thrs();

t1.start();

t2.start();

t3.start();

}

}

**Output:**

enter the value of n

3

The value of i is: 3

The value of i is: 2

The value of i is: 1

enter the value of n

5

The value of i is: 5

The value of i is: 4

The value of i is: 3

The value of i is: 2

The value of i is: 1

enter the value of n

4

The value of i is: 4

The value of i is: 3

The value of i is: 2

The value of i is: 1

**Q4. SYNCHRONIZATION**

Presidency University is organizing an event. While each participant arrives, get the name of the participant and assign participant IDs to each participant starting from 101, and should be following with sequence of ID for the further participants. Presidency has created a String array that has the following info.

{“Hi”, “name of the participant”, “ID of the participant”, “Welcome Message”}. The registration team register the Participant details. You can create a class “Participant” with participantId, participantName as members. Create a setter method to set the participant details. Once the participants are registered, the welcome message is printed by separate Threads on Participant instance. Write a java program to handle this scenario.package testthread;

**Aim:**

The aim of this program is to simulate an event registration process at Presidency University. As participants arrive, the program collects their names and assigns unique participant IDs starting from 101. After registration, separate threads are used to print a welcome message for each participant.

**Algorithm:**

1. Create a class named `Presidency` with a static synchronized method `display(int id, String name)`:

a. Define a string array named `welcomeMessage` containing a welcome message with placeholders for the participant's name and ID.

b. Loop through the `welcomeMessage` array.

c. Inside the loop:

- Simulate a delay using `Thread.sleep(1000)` to create a one-second interval between message elements.

- Print each element of the welcome message.

2. Create a class named `Participant` that extends the `Thread` class:

a. Declare a static integer variable `temp` and initialize it to 100. This variable will be used to assign unique participant IDs.

b. Declare instance variables: `participantId`, `participantName`, and a `Scanner` object named `scan`.

c. Create a method named `setDetails()` to collect the participant's name from the user and assign a unique participant ID.

d. Implement the `run()` method:

- Call the `Presidency.display(participantId, participantName)` method to display the welcome message for the participant.

3. In the `PUEvent` class (the main class):

a. Create three instances of the `Participant` class: `p1`, `p2`, and `p3`.

b. Call the `setDetails()` method for each participant to collect their names and assign unique IDs.

c. Start each participant thread using the `start()` method.

**Program Explanation:**

- This program simulates the event registration process at Presidency University, where participants are assigned unique IDs and receive a welcome message.

- The `Presidency` class contains a static synchronized method `display(int id, String name)` that displays a welcome message for each participant. It uses a string array with message elements and introduces a one-second delay between each message element.

- The `Participant` class extends the `Thread` class. It assigns unique participant IDs starting from 101 and collects the participant's name. The `run()` method is implemented to display the welcome message using the `Presidency.display()` method.

- In the `PUEvent` class (the main class), three participant instances are created: `p1`, `p2`, and `p3`. For each participant, the `setDetails()` method is called to collect their name and assign a unique ID. Each participant is started as a separate thread.

- As a result, when the program is executed, it registers the participants, assigns unique IDs, and prints welcome messages for each participant. The use of synchronized methods ensures that the messages are displayed in the desired sequence.

**Program:**

**package** scenario3;

**import** java.util.Scanner;

**class** Presidency{

**synchronized** **static** **void** display(**int** id, String name){

String welcomeMessgae[] ={"Hi",name, " yourId "+ id+"", " welcome to Presidency University"};

**for**(**int** i=0; i<welcomeMessgae.length; i++){ **try** {

Thread.*sleep*(1000); System.***out***.println(welcomeMessgae[i]);

}**catch**(Exception e){

}

}

}

}

**class** Participant **extends** Thread{ **static** **int** *temp* = 100;

**int** participantId;

String participantName;

Scanner scan = **new** Scanner(System.***in***); **public** **void** setDetails(){

System.***out***.println("Enter the name"); participantName = scan.nextLine(); participantId = ++*temp*;

}

**public** **void** run(){

Presidency.*display*(participantId, participantName);

}

}

**public** **class** PUEvent {

**public** **static** **void** main(String[] args) { Participant p1 = **new** Participant(); Participant p2 = **new** Participant(); Participant p3 = **new** Participant(); p1.setDetails();

p2.setDetails(); p3.setDetails(); p1.start();

p2.start();

p3.start();

}

}

**Output:**

Enter the name

prakash

Enter the name

ravi

Enter the name

raju

Hi

prakash

yourId 101

welcome to Presidency University

Hi

raju

yourId 103

welcome to Presidency University

Hi

ravi

yourId 102

welcome to Presidency University

**Q5. SYNCHRONIZATION**

Implment Thread Synchronization for a given resource to avoid race condition to perform the following operations:

i. Create a Resource class to keep two resources [ and ]. No thread can take ] without [ .

ii. Create three threads with the names “Hello”, “Synchronized” and “Block” to access the above resource without synchronization.

iii. Access the above resource using synchronization.

**Aim:**

The aim of the Java program is to create multiple threads that access a synchronized static method `display` of the `exam1` class. This program allows the user to input an integer 'n', and then each thread counts down from 'n' to 1 with a 1-second delay between each count. The program demonstrates the use of synchronization in a static method.

**Algorithm:**

1. Create a class `exam1` with a synchronized static method `display`. This method takes input for an integer 'n' and then counts down from 'n' to 1, printing each value with a 1-second delay between each count.

2. Create a class `Thrs` that extends the `Thread` class. In its `run` method, call the synchronized `display` method from the `exam1` class.

3. In the `quiz` class (the main class), create three `Thrs` objects, representing three threads.

4. Start all three threads by invoking their `start` methods.

**Program Explanation:**

- The program starts by creating a class named `exam1` with a synchronized static method `display`. This method reads an integer 'n' from the user and counts down from 'n' to 1, printing each value with a 1-second delay between each count.

- The `Thrs` class is defined, extending the `Thread` class. In its `run` method, it calls the synchronized `display` method from the `exam1` class. This means multiple threads can access the method, but only one thread can execute it at a time due to the `synchronized` keyword.

- In the `quiz` class (the main class), three `Thrs` objects, namely `t1`, `t2`, and `t3`, are created.

- The `start` method is called on each thread, which initiates the execution of the `run` method in each thread.

- The program demonstrates how synchronization works with a static method to ensure that only one thread at a time can execute the `display` method, preventing concurrent access that could lead to unexpected behavior.

Overall, this program showcases the use of synchronized methods and multi-threading in Java.

**Program:**

**//Without Synchronization**

**package** pp5;

**class** Resource {

**void** use(String name) {

System.***out***.print("[" + name);

**try** {

Thread.*sleep*(1000);

} **catch**(InterruptedException e) {

System.***out***.println("Interrupted");

}

System.***out***.println("]");

}

}

**class** MyThread **extends** Thread {

String name;

Resource r;

MyThread (String name,Resource r){

**super**(name);

**this**.name = name;

**this**.r=r;

}

**public** **void** run() {

//synchronized(r) {

r.use(name);

}

}

//} closed brace of synchronized

**public** **class** TestMultiThread {

**public** **static** **void** main(String args[]) {

Resource res=**new** Resource();

MyThread t1=**new** MyThread("Hello",res);

MyThread t2=**new** MyThread("Synchronized",res);

MyThread t3=**new** MyThread("Block",res);

t1.start();

t2.start();

t3.start();

**try** {

t1.join();

t2.join();

t3.join();

} **catch** (InterruptedException excetion) {

System.***out***.println("Inturruption occurs in Main Thread");

}

}

}

**Output:**

[Synchronized[Block[Hello]

]

]

**//With Synchronization**

**package** pp5;

**class** Resource {

**void** use(String name) {

System.***out***.print("[" + name);

**try** {

Thread.*sleep*(1000);

} **catch**(InterruptedException e) {

System.***out***.println("Interrupted");

}

System.***out***.println("]");

}

}

**class** MyThread **extends** Thread {

String name;

Resource r;

MyThread (String name,Resource r){

**super**(name);

**this**.name = name;

**this**.r=r;

}

**public** **void** run() {

**synchronized**(r) {

r.use(name);

}

}

}

**public** **class** TestMultiThread {

**public** **static** **void** main(String args[]) {

Resource res=**new** Resource();

MyThread t1=**new** MyThread("Hello",res);

MyThread t2=**new** MyThread("Synchronized",res);

MyThread t3=**new** MyThread("Block",res);

t1.start();

t2.start();

t3.start();

**try** {

t1.join();

t2.join();

t3.join();

} **catch** (InterruptedException excetion) {

System.***out***.println("Inturruption occurs in Main Thread");

}

}

}

**Output:**

[Hello]

[Synchronized]

[Block]

**Q6. MULTI THREAD**

With the help of Multithreading concept, create a thread titled "primeThread" by implementing Runnable interface. "primeThread" will be responsible to print all the prime numbers from 2 to 100 in a regular interval of 0.5 seconds.

On executing this application, get the following info also and print the same.

a. Get the id, name and priority of the main thread

b. Change the name and priority of the main thread

c. Use isAlive method to check the status of the child thread.

**Program:**

**package** scenario6;

**class** Prime **extends** Thread{

**boolean** isPrime = **true**;

**public** **void** run()

{

**for**(**int** num=2;num<100;num++)

{

**for**(**int** div=2;div<=num/2;div++)

{

**if**(num%div==0)

{

isPrime=**false**;

}

}

**if**(isPrime)

{

**try**

{

System.***out***.println("Prime :"+ num);

Thread.*sleep*(500);

}

**catch**(InterruptedException ie)

{

}

}

isPrime=**true**;

}

}

}

**public** **class** PrimeNumberMultiThread {

**public** **static** **void** main(String[] args) {

//get the name and Id of the main thread

Thread mainThread = Thread.*currentThread*();

System.***out***.println("Id of Main Thread: "+ mainThread.getId());

System.***out***.println("Name of main Thread: "+ mainThread.getName());

System.***out***.println("Default Priority of Main Thread: "+ mainThread.getPriority());

//Change the name and priority of the main thread and print the same.

mainThread.setName("ThreadMainClass");

mainThread.setPriority(Thread.***MIN\_PRIORITY***);

System.***out***.println("Name of main Thread after changing: "+mainThread.getName()); System.***out***.println("Priority of main thread after changing: "+mainThread.getPriority());

//creating a child thread "primeThread"

Prime prime= **new** Prime();

prime.setName("primeThread");

prime.start();

//Thread grouop of the child-Thread

System.***out***.println("child-Thread1: "+ prime);

//checking with isAlive() method

System.***out***.println("primeThread is Alive??: "+ prime.isAlive());

}

}

**Output:**

Id of Main Thread: 1

Name of main Thread: main

Default Priority of Main Thread: 5

Name of main Thread after changing: ThreadMainClass

Priority of main thread after changing: 1

child-Thread1: Thread[primeThread,1,main]

primeThread is Alive??: true

Prime :2

Prime :3

Prime :5

Prime :7

Prime :11

Prime :13

Prime :17

Prime :19

Prime :23

Prime :29

Prime :31

Prime :37

Prime :41

Prime :43

Prime :47

Prime :53

Prime :59

Prime :61

Prime :67

Prime :71

Prime :73

Prime :79

Prime :83

Prime :89

Prime :97

**Q7. MULTI THREAD**

**With the help of Multithreading concept, create a thread titled "fiboThread" by extending the Thread class. "fiboThread" will be responsible to print fibonacci series of first 20 numbers, in a regular interval of 0.25 seconds.**

**On executing this application, get the following info also and print the same.**

**a. Get the id, name and priority of the main thread**

**b. Change the name and priority of the main thread**

**c. Use isAlive method to check the status of the child thread.**

**Aim:**

The aim of the Java program is to create multiple threads that access a synchronized static method `display` of the `exam1` class. This program allows the user to input an integer 'n', and then each thread counts down from 'n' to 1 with a 1-second delay between each count. The program demonstrates the use of synchronization in a static method.

**Algorithm:**

1. Create a class `exam1` with a synchronized static method `display`. This method takes input for an integer 'n' and then counts down from 'n' to 1, printing each value with a 1-second delay between each count.

2. Create a class `Thrs` that extends the `Thread` class. In its `run` method, call the synchronized `display` method from the `exam1` class.

3. In the `quiz` class (the main class), create three `Thrs` objects, representing three threads.

4. Start all three threads by invoking their `start` methods.

**Program Explanation:**

- The program starts by creating a class named `exam1` with a synchronized static method `display`. This method reads an integer 'n' from the user and counts down from 'n' to 1, printing each value with a 1-second delay between each count.

- The `Thrs` class is defined, extending the `Thread` class. In its `run` method, it calls the synchronized `display` method from the `exam1` class. This means multiple threads can access the method, but only one thread can execute it at a time due to the `synchronized` keyword.

- In the `quiz` class (the main class), three `Thrs` objects, namely `t1`, `t2`, and `t3`, are created.

- The `start` method is called on each thread, which initiates the execution of the `run` method in each thread.

- The program demonstrates how synchronization works with a static method to ensure that only one thread at a time can execute the `display` method, preventing concurrent access that could lead to unexpected behavior.

Overall, this program showcases the use of synchronized methods and multi-threading in Java.

**Program:**

**package** scenario7;

**class** Fibonacci **implements** Runnable

{

**int** n1 = 0, n2=1;

**final** **int** MAX =7;

**public** **void** run()

{

System.***out***.println("Fib: "+n1);

System.***out***.println("Fib: "+n2);

**for**(**int** i=2;i<MAX;i++){ **int** n3 = n1+n2;

**try**{

System.***out***.println("Fib: "+ n3);

Thread.*sleep*(250);

}**catch**(InterruptedException ie){

}

n1=n2;

n2=n3;

}

}

}

**public** **class** FiboNumberMultiThread {

**public** **static** **void** main(String[] args) {

//get the name and Id of the main thread

Thread mainThread = Thread.*currentThread*();

System.***out***.println("Id of Main Thread: "+ mainThread.getId());

System.***out***.println("Name of main Thread: "+ mainThread.getName());

System.***out***.println("Default Priority of Main Thread: "+ mainThread.getPriority());

//Change the name and priority of the main thread and print the same.

mainThread.setName("ThreadMainClass");

mainThread.setPriority(Thread.***MIN\_PRIORITY***);

System.***out***.println("Name of main Thread after changing: "+mainThread.getName()); System.***out***.println("Priority of main thread after changing: "+mainThread.getPriority());

//creating a child thread "fiboThread"

Fibonacci fibonacci = **new** Fibonacci();

Thread fib = **new** Thread(fibonacci,"fiboThread"); fib.start();

//Thread grouop of the child-Thread

System.***out***.println("child-Thread2: "+fib);

//checking with isAlive() method

System.***out***.println("primeThread is Alive??: "+ fib.isAlive());

}

}

**Output:**

Id of Main Thread: 1

Name of main Thread: main

Default Priority of Main Thread: 5

Name of main Thread after changing: ThreadMainClass

Priority of main thread after changing: 1

child-Thread2: Thread[fiboThread,1,main]

primeThread is Alive??: true

Fib: 0

Fib: 1

Fib: 1

Fib: 2

Fib: 3

Fib: 5

Fib: 8

**Q6. INTER THREAD COMMUNICATION**

Write a simulation program for the fruit market. The farmer will be able to produce different types of fruits (apple, orange, grape, and watermelon), and put them in the market to sell. The market has limited capacity and farmers have to stand in a queue. The farmer can sell 10 fruits at a time to the consumer. So the farmer should have fruits in multiple of 10’s. Consumers can come to the market any time and purchase their desired fruits; and if the fruits they want to buy runs out, they are willing to wait until the supply of that kind is ready.

**Aim:**

The aim of this program is to simulate a fruit market where farmers can produce different types of fruits and put them up for sale. Consumers can purchase fruits in multiples of 10, and if a specific type of fruit runs out of stock, they are willing to wait until it is available.

**Algorithm:**

1. Create a class named `MarketQueue` with the following members:

- `typeOfFruit`: A string to represent the type of fruit available.

- `order`: A boolean variable to control the order of interaction between farmers and consumers.

2. Implement a synchronized `put(String typeOfFruit)` method in the `MarketQueue` class:

a. Check if `order` is `false`, indicating that it's the farmer's turn. If not, wait for the consumer's turn.

b. Set the `typeOfFruit` with the provided fruit name.

c. Update `order` to `false` to indicate that it's the consumer's turn.

d. Print the available fruit and notify any waiting consumers.

3. Implement a synchronized `get()` method in the `MarketQueue` class:

a. Check if `order` is `true`, indicating that it's the consumer's turn. If not, wait for the farmer's turn.

b. Update `order` to `true` to indicate that it's the farmer's turn.

c. Print the received fruit and notify any waiting farmers.

4. Create a class named `Farmer` that implements the `Runnable` interface:

a. Initialize `marketQueue` to interact with the market.

b. Collect the type of fruit and the number of items to sell from the farmer.

c. Check if the number of items is a multiple of 10.

d. If it's a multiple of 10, put each batch of 10 items into the market using the `marketQueue.put()` method.

e. If the number of items is not a multiple of 10, exit the program.

5. Create a class named `Consumer` that implements the `Runnable` interface:

a. Initialize `marketQueue` to interact with the market.

b. In an infinite loop, continuously call the `marketQueue.get()` method to purchase fruits.

6. In the `InterThreadMarket` class (the main class):

a. Create an instance of `MarketQueue` named `mq`.

b. Create an instance of `Farmer` and `Consumer` to represent market participants.

**Program Explanation:**

- This program simulates a fruit market where farmers can produce and put up fruits for sale, and consumers can purchase them in multiples of 10.

- The `MarketQueue` class provides a synchronized mechanism to control the order of interaction between farmers and consumers.

- Farmers enter the type of fruit they have and the number of items to sell. If the number of items is a multiple of 10, they put the fruits into the market using the `marketQueue.put()` method.

- Consumers continuously purchase fruits in multiples of 10 using the `marketQueue.get()` method.

- If the number of items a farmer wants to sell is not a multiple of 10, the program exits with a message to try again in the queue.

- This program demonstrates how multithreading can be used to simulate interactions between farmers and consumers in a fruit market. It ensures that consumers are willing to wait for specific fruits to become available.

**Program:**

**package** scenario4;

**import** java.util.Scanner;

**class** MarketQueue

{

String typeOfFruit;

**boolean** order = **true**;

**public** **synchronized** **void** put(String typeOfFruit)

{

**if**(order == **false**){

**try**{

wait();

}**catch**(InterruptedException ie){}

}

**this**.typeOfFruit = typeOfFruit;

order = **false**;

System.***out***.println("Fruit available : " +typeOfFruit); notify();

}

**public** **synchronized** **void** get()

{

**if**(order == **true**)

{

**try**{

wait();

}

**catch**(InterruptedException ie){}

}

order = **true**;

System.***out***.println("Received Fruit : " + typeOfFruit); notify();

}

}

**class** Farmer **implements** Runnable

{

MarketQueue marketQueue;

**int** numOfFruits;

Scanner scan = **new** Scanner(System.***in***);

**public** Farmer(MarketQueue marketQueue)

{

**this**.marketQueue= marketQueue;

**new** Thread(**this**,"Farmer").start();

}

**public** **void** run()

{

System.***out***.println("Type of the fruit that Farmer has?");

String typeOfFruit = scan.nextLine(); System.***out***.println("Number of itmes (in multiplies of 10)"); numOfFruits = scan.nextInt();

**if**(numOfFruits%10==0)

{

**for**(**int** i=1;i<numOfFruits;i=i+10)

{

marketQueue.put(typeOfFruit+" "+(i)+" to "+(i+9));

}

}

**else**

{

System.***out***.println("Try again in queue");

System.*exit*(0);

}

}

}

**class** Consumer **implements** Runnable

{

MarketQueue marketQueue;

**public** Consumer(MarketQueue marketQueue)

{

**this**.marketQueue = marketQueue;

**new** Thread(**this**,"Farmer").start();

}

**public** **void** run()

{

**while**(**true**)

{

marketQueue.get();

}

}

}

**public** **class** InterThreadMarket

{

**public** **static** **void** main(String[] args)

{

MarketQueue mq = **new** MarketQueue();

Farmer farmer = **new** Farmer(mq);

Consumer consumer = **new** Consumer(mq);

}

}

**Output:**

Sample1

Type of the fruit that Farmer has?

apple

Number of itmes (in multiplies of 10)

30

Fruit available : apple 1 to 10

Received Fruit : apple 1 to 10

Fruit available : apple 11 to 20

Received Fruit : apple 11 to 20

Fruit available : apple 21 to 30

Received Fruit : apple 21 to 30

Sample2

Type of the fruit that Farmer has?

orange

Number of itmes (in multiplies of 10)

70

Fruit available : orange 1 to 10

Received Fruit : orange 1 to 10

Fruit available : orange 11 to 20

Received Fruit : orange 11 to 20

Fruit available : orange 21 to 30

Received Fruit : orange 21 to 30

Fruit available : orange 31 to 40

Received Fruit : orange 31 to 40

Fruit available : orange 41 to 50

Received Fruit : orange 41 to 50

Fruit available : orange 51 to 60

Received Fruit : orange 51 to 60

Fruit available : orange 61 to 70

Received Fruit : orange 61 to 70